CS 261: Data Structures

Week 2: Dictionaries and hash tables

Lecture 2b: Collision resolution

David Eppstein
University of California, Irvine

Spring Quarter, 2025

Collisions and collision resolution

Birthday paradox

A class of 50 students has in expectation $\binom{50}{2}\frac{1}{365} \approx 3.35$ pairs of students who both have the same birthday

The sum comes from linearity of expectation:

- There are $\binom{50}{2}$ pairs of students
- Each pair has the same birthday with probability $\frac{1}{365}$
- Expected number of events is sum of their probabilities

Random functions have many collisions

Suppose we map n keys randomly to a hash table with N cells.

Then expected number of pairs of keys that collide is $\binom{n}{2}/N \approx n^2/(2N)$

We get significant chance of collisions when # cells $\leq$ # keys$^2/2$

Conclusion: Avoiding all collisions with random functions needs hash table size $\Omega(n^2)$, way too big to be efficient

Resolving collisions: Hash chaining (bucketing)

Instead of storing a single key–value pair in each array cell,
store an association list (collection of key–value pairs)

To set or get value for key k, search the list in $A[h(k)]$ only

Time will be fast enough if these lists are small

(But in practice the overhead of having a multi-level data structure and keeping a list per array cell means the constant factors in time and space bounds are larger than other methods.)

First hashing method known,
generally credited to IBM researcher Peter Luhn, 1953

Expected analysis of hash chaining

Assumption: N cells with load factor $\alpha = n/N = O(1)$

(Use dynamic tables and increase the table size when $N << n$)

Then, for any key k, the time to set or get the value for k is $O(\ell_k)$, the number of key–value pairs in $A[h(k)]$.

By linearity of expectation,

$$E[\ell_k] = 1 + \sum_{\text{key } j \neq k} \Pr[j \text{ collides with } k] = \frac{n-1}{N} < \alpha = O(1).$$

So with tables of this size, expected time/operation is $O(1)$.

Expected size of the biggest chain

A random or arbitrary key has expected time per operation $O(1)$

But what if attacker chooses key whose cell has max # keys?

(Again, assuming constant load factor)

Chernoff bound: The probability that any given cell has $\geq x\alpha$ keys is at most

$$\left(\frac{e^{x-1}}{x^x}\right)^{\alpha}$$

(and is lower-bounded by a similar expression)

For $x = C\frac{\log n}{\log\log n}$, simplifies to $1/N^c$ for some c (depending on C)

- $c > 1 \Rightarrow$ with high probability all cells $\leq x\alpha$ (union bound)
- $c < 1 \Rightarrow$ expect many cells $\geq x\alpha$ (linearity of expectation)

Expected size of largest cell is $\Theta(\frac{\log n}{\log\log n})$

Resolving collisions: Open addressing

Instead of having a single array cell that each key can go to,
use a "probe sequence" of multiple cells

Look for the key at each position of the probe sequence until either:

You find a cell containing the key

You find an empty cell (and deduce that key is not present)

Many variations including linear probing and cuckoo hashing (today)

Linear probing

What is linear probing?

The simplest possible probe sequence:
Look for key k in cells $h(k), h(k)+1, h(k)+2, \ldots \pmod{N}$

Invented by Gene Amdahl, Elaine M. McGraw, and Arthur Samuel, 1954, and first analyzed by Donald Knuth, 1963

- Fast in practice: simple lookups and insertions, works well with cached memory
- Commonly used
- Requires a high-quality hash function (next time)
- Load factor α must be < 1 (else no room for all keys)

Lookups and insertions

With array A of length N, holding n keys, and hash function h:

To look up value for key k:

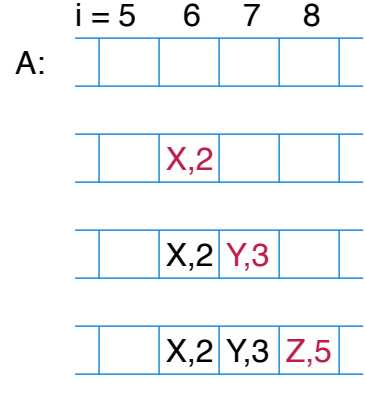
```
i = h(k)
while A[i] has wrong key:
    i = (i + 1) % N
if A[i] is empty:
    raise exception
return value from A[i]
```

To set value for k to x:

```
i = h(k)
while A[i] has wrong key:
    i = (i + 1) % N
if A[i] is empty:
    n += 1
    if n/N too large:
        expand A and rebuild
A[i] = k,x
```

Example

key	value	hash
X	2	6
Y	3	7
Z	5	6



Default position for Z was filled $\Rightarrow$ placed at the next available cell

Have to be careful with deletion: blanking the cell for X would make the lookup algorithm think Z is also missing!

Deletion

First, find key and blank cell:

```
i = h(k)
while A[i] has wrong key:
    i = (i + 1) % N
if A[i] is empty:
    raise exception
A[i] = empty
```

Then, pull other keys forward:

```
j = (i + 1) % N
while A[j] nonempty:
    k = key in A[j]
    if h(k) <= i < j (mod N):
        A[i] = A[j]
        A[j] = empty
        i = j
    j = (j + 1) % N
```

Result = As if remaining keys inserted without the deleted key

Blocks of nonempty cells

The analysis is controlled by the lengths of contiguous blocks of nonempty cells (with empty cells at both ends)

Any sequence of B cells is a block only when:

- Nothing hashes to the empty cells at both ends
- Exactly B items hash to somewhere inside
- For each i, at least i items hash to the first i cells

Our analysis will only use the middle condition:
Exactly B items hash into the block

Probability of forming a block

Suppose we have a sequence of exactly B cells

Expected number of keys hashing into it: αB

To be a block, actual number = B = $\frac{1}{\alpha}$expected

Chernoff bound:
Probability this happens is at most

$$\left(\frac{e^{1/\alpha-1}}{(1/\alpha)^{1/\alpha}}\right)^{\alpha B} = c^B$$

for some constant $c < 1$ depending on α

Long sequences of cells are exponentially unlikely to form blocks!

Linear probing analysis

Expected time per operation on key k

$= O(\text{expected length of block containing } h(k))$

$= \sum_{\text{possible blocks}} \Pr[\text{it is a block}] \times \text{length}$

$= \sum_{\ell} (\text{number of blocks of length } \ell \text{ containing } k) \times c^{\ell} \times \ell$

$= \sum_{\ell} \ell \times c^{\ell} \times \ell$

$= O(1)$

(because the exponentially-small c^{ℓ} overwhelms the factors of ℓ)

Expected length of the longest block

A random or arbitrary key has expected time per operation $O(1)$

But what if attacker chooses key whose block has max # length?

Same analysis $\Rightarrow$ blocks of length $C \log n$ have probability $1/N^{\Theta(1)}$ (inverse-exponential in length) of existing

- Large $C \Rightarrow$ high probability of no blocks bigger than $C \log n$
- Small $C \Rightarrow$ expect many blocks bigger than $C \log n$

Can prove: Expected size of largest block is $\Theta(\log n)$

Cuckoo hashing

Cuckoo

Cuckoo = bird that lays eggs in other birds' nests

when it hatches, the baby pushes other eggs and chicks out

Main ideas

Cuckoo hashing = open addressing, only two possible cells per key

[Pagh and Rodler 2001]

Lookup: always $O(1)$ time – just look in those two cells

Deletion: easy, just blank your cell

Insertion: if a key is already in your cell, push it out
(and recursively insert it into its other cell)

Comparison with other methods

When an adversary can pick a bad key, other methods can be slow:

- $\Theta(\log n / \log \log n)$ for hash chaining
- $\Theta(\log n)$ for linear probing

In contrast, cuckoo hash lookup is always $O(1)$

Cuckoo hashing is useful when keys can be adversarial
or when lookups must always be fast

- e.g. internet packet filter

But for fast average case, linear probing may be better

Requirements and variations

Sometimes the two cells for each key are in two different arrays but it works equally well for both to be in a single array

(We will use the single-array version)

Basic method needs load factor $\alpha < 1/2$ to avoid infinite recursion

More complicated variations can handle any $\alpha < 1$

- Store a fixed number of keys per cell, not just one
- Keep a "stash" of $O(1)$ keys that don't fit

Easy operations: Lookup and delete

With array A, hash functions $h1$ and $h2$ such that $h1(k) \neq h2(k)$:

To look up value for key k:

```
if A[h1(k)] contains k:
    return value in A[h1(k)]
if A[h2(k)] contains k:
    return value in A[h2(k)]
raise exception
```

To delete key k:

```
if A[h1(k)] contains k:
    empty A[h1(k)] and return
if A[h2(k)] contains k:
    empty A[h2(k)] and return
raise exception
```

Insertion

To insert key k with value x:

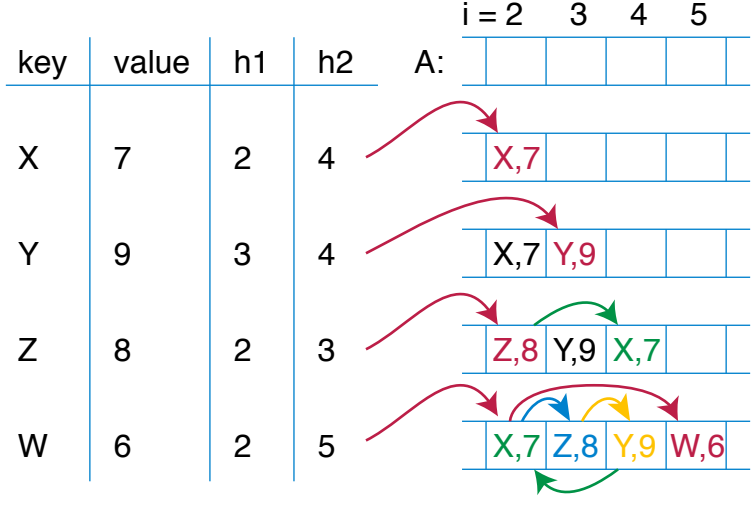
```
i = h1(k)
while A[i] is non-empty:
    swap k,x with A[i]
    i = h1(k) + h2(k) - i
A[i] = k,x
```

Not shown: test for infinite loop

If we ever return to a state where we are trying to re-insert the original key k into its original cell $h1(k)$, the insertion fails

(put k into the stash or rebuild the whole data structure)

Example


key
value
h1
h2
X
7
2
4
Y
9
3
4
Z
8
2
3
W
6
2
5
A:
i = 2
3
4
5
X,7
X,7
Y,9
Z,8
Y,9
X,7
X,7
Z,8
Y,9
W,6

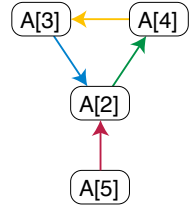
Graph view of cuckoo state

Vertices = array cells

Edges = pairs of cells used by the same key
Directed away from cell where the key is stored

key	value	h1	h2
X	7	2	4
Y	9	3	4
Z	8	2	3
W	6	2	5

i = 2	3	4	5
A: X,7	Z,8	Y,9	W,6



Key properties: Each vertex has at most one outgoing edge
Each component is a pseudotree (cycle with trees attached)

Graph view of key insertion

Add edge $h1(k) \rightarrow h2(k)$ to the graph

If $h1(k)$ already had an outgoing edge:

Follow path of outgoing edges, reversing each edge in the path

Case analysis:

- $h1(k)$ was in a tree: path stops at previous root
- $h1(k)$ was in a pseudotree and $h2(k)$ was in a tree: path loops through cycle, comes back out, reverses new edge, and stops at root of tree
- $h1(k)$ and $h2(k)$ are both in pseudotrees (or same pseudotree): fails, too many edges for all vertices to have only one outgoing edge

Summary of graph interpretation

The insertion algorithm

- Works correctly when the undirected graph with edges $h1(k)$—$h2(k)$ has at most one cycle per component
- Fails when any component has two cycles
  (equivalently: more edges than vertices)
- Takes time proportional to the component size

The graph is random (each two vertices equally likely to be edge)

We know a lot about random graphs!

For n vertices and αn edges, $\alpha < 1/2$, with high probability:

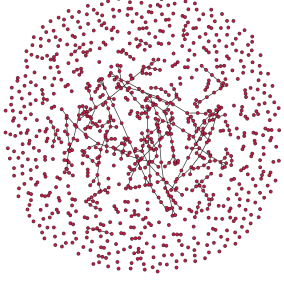
- All components are trees or pseudotrees
- Pr[component of v has size ℓ] is exponentially small in ℓ
- Expected size of component containing v is $O(1)$
- But expected largest component size $\Theta(\log n)$
- E[number of pseudotrees] $= O(1)$

For $\alpha = 1/2$:

- Max component size $\Theta(n^{2/3})$
- May have > 1 big component

For $\alpha > 1/2$:

- One giant component, $\Theta(n)$
- Rest: small trees and pseudotrees



What random graphs imply for cuckoo hashing

For load factor $\alpha < 1/2$:

- With high probability, all keys can be inserted
- Expected time per insertion is $O(1)$
- Expected time of slowest insertion is $O(\log n)$

For load factor $\alpha \geq 1/2$:

- It doesn't work (in simplest variation)

Summary

Summary

- Dictionary problem: Updating and looking up key–value pairs
- In standard libraries (e.g. Java HashMap, Python dict)
- Hash tables: direct addressing + hash function
- Hash functions: perfect, cryptographic, random, k-independent (algebraic and tabulation)
- Hash chaining, expected time, expected worst case
- Linear probing, expected time, expected worst case
- Cuckoo hashing and random graphs
- Reviewed basic probability theory + Chernoff bound

References and image credits

Ed g2s. Birthday candles. CC-BY-SA image, February 13 2005. URL `https://commons.wikimedia.org/wiki/File:Birthday_candles.jpg`.

Rasmus Pagh and Flemming Friche Rodler. Cuckoo hashing. In Friedhelm Meyer auf der Heide, editor, *Algorithms – ESA 2001, 9th Annual European Symposium, Aarhus, Denmark, August 28–31, 2001, Proceedings*, volume 2161 of *Lecture Notes in Computer Science*, pages 121–133. Springer, 2001. doi: 10.1007/3-540-44676-1_10.

Chris Romeiks. Common cuckoo (Cuculus canorus). Cropped by Bogbumper and MPF from CC-BY-SA image, May 2 2010. URL `https://commons.wikimedia.org/wiki/File:Cuculus_canorus_vogelartinfo_chris_romeiks_CHR0791_cropped.jpg`.